© *Jean-Baptiste APOUNG KAMGA <jean-baptiste.apoung@math.u-psud.fr>*

Fiche de TP3 : Programmation générique en C++

Le but de ce TP est d'aborder les outils fournis par le C++ , pour la programmation générique. Pour aller vite dans la saisie des bouts de programmes fournis dans les exercices, nous rappelons que les fichiers sources de ces programmes sont téléchargeables à l'adresse **/doc/apoung/M2_IM_STAT**

Thème - 1 *Généralités*

Exercice-1 : Ce programme compile-t-il ? justifier.

Listing 1 – instanciation.cpp

```
#include <iostream>
//-----------------------------------------------
template<typename T> class C{
public:
static int geti(const T& t){return t.i();}
double getd(const T& t){return t.d;}
};
//-----------------------------------------------
struct E{
int i()const{return 1;}
};
//-----------------------------------------------
struct F{double d;};
//-----------------------------------------------
int main(){
E e;
std::cout << C<E>::geti(e) << std::endl;
F f;
f.d = 3.14;
C<F> cf;
std::cout << cf.getd(f) << std::endl;
return 0;
}
```

Exercice-2 : Qu'affiche le programme suivant

Listing 2 – spécialisation.cpp

```cpp
#include <iostream>
template<typename T>
class C {
public:
void print(){std::cout << "C<T>" << std::endl;}
};
//-----------------------------------------------
template<typename T> class C<T*>{
public:
void print(){std::cout << "C<T*>" << std::endl;}
};
//-----------------------------------------------
template<> class C<int>{
public:
void print(){std::cout << "C<int>" << std::endl;}
};
//-----------------------------------------------
int main(){
C<double> cd; cd.print();
C<int> ci; ci.print();
C<int*> cpi; cpi.print();
C<int**> cppi; cppi.print();
return 0;
}
```

Exercice-3 : Expliquer ce que fait le programme suivant

Listing 3 – deduction.cpp

```cpp
#include <iostream>
 using namespace std;
 //-----------------------------------------------------
 template<typename T1, typename T2, int S>
 void f(T1 t1, T2 (&t2)[S]){
 cout<< t1 << " -- " << t2[0] << " -- " << S << endl;
 }
 //-----------------------------------------------------
 class C {
 private:
 friend ostream& operator << (ostream& os, const C& c){
 os << "C"; return os;
 }
 };
 //-----------------------------------------------------
 int main(int argc, char** argc){
 f(2, "abcde");
 C tc[5];
 f('s',tc);
 return 0;
 }
```

Exercice-4 : Expliquer ce qu'affiche le programme suivante. Justifier.

Listing 4 – resolution_fonction.cpp

```
#include <iostream>
using namespace std;
//-------------------------------------------------------
void f(char c){std::cout<<" appel de char \n";}
//-------------------------------------------------------
template<typename T>
void g(T t){
f(1);
}
//-------------------------------------------------------
void f(int i){std::cout<<" appel de int \n";}
//-------------------------------------------------------
int main(){
 g(2);
return 0;
}
```

Exercice-5 : Expliquer ce qu'affiche le programme suivante. Justifier.

Listing 5 – resolution_types.cpp

```
#include <iostream>
#include <typeinfo>
using namespace std;
//-------------------------------------------------------
namespace {
const double E =1;
const double pe1 = 1;
const double pd1 = 1;
}
//-------------------------------------------------------
class C;
template < class T>
class A{
  class B;
public:
  void algorithme1(){
     C* pc;  T* pt;   A* pa;   B* pb;
     E* pe1;
    typename T::D* pd1;
    std::cout<<"Dans " << __FUNCTION__
             <<" pd1  est du type  " <<typeid(pd1).name()<<"\n";
    std::cout<<"Dans " << __FUNCTION__
             <<" pe1  est du type  " <<typeid(pe1).name()<<"\n";
  }
  void algorithme2(){
    class E;
    C* pc;  T* pt;  A* pa;  B* pb;
    E* pe1;
    T::D* pd1;
    std::cout<<"Dans " << __FUNCTION__
             <<" pd1  est du type  " <<typeid(pd1).name()<<"\n";
    std::cout<<"Dans " << __FUNCTION__
```

```
            <<" pe1  est du type  " <<typeid(pe1).name()<<"\n";
  }
};
//-------------------------------------------------------
class H {
public:
  typedef double D;
};
//-------------------------------------------------------
class L {
public:
  static double D;
};
double L::D = 0.0;
//////////g++ resolution_types.cpp -o resolution_types
int main(){
 A<H> ah; ah.algorithme1();
 A<L> al; al.algorithme2();
return 0;
}
```

Exercice-6 : Expliquer pourquoi ce code ne compile pas et corriger si possible.

Listing 6 – template_friend_template.cpp

```
#include <iostream>
template<int N>
class A{
public:
template<int M>
friend void g(const A<M>& a){
  std::cout<<" call g(const A<"<< M <<">&)\n";
}
};
//-------------------------------------------------
int main(int argc, char** argv){
  A<2> a2;
  A<3> a3;
  g(a2);
  g(a3);
  return 0;
}
```

Thème - 2 *Classes génériques : Applications*

Exercice-1 : On souhaite disposer d'un tableau générique de taille fournie pendant la compilation pouvant contenir une collection d'objets de type fourni aussi à la compilation. On convient alors

de définir une classe générique
`template class TinyVector<typenamt T, int N>`

Q-1-1 : Définir et implémenter ce type. En fournissant en autres les opérateurs **`[]`** pour accéder aux données de manière sécurisée et chevron **`<<`** pour afficher un TinyVector sur la console.

Q-1-2 : Est-il raisonnable de munir cette classe des opérations arithmétiques usuels ?

Exercice-2 : Cet exercice porte sur l'écriture d'une classe **`Tableau`** extensible, dont la vocation sera de stocker des données avec accès direct de manière contiguë. On pourra **sans perte de données** lui ajouter ou lui supprimer des éléments.

Cette classe fournira :

Cycle de vie
- Un constructeur utilisant en argument optionnel la capacité du tableau (par défaut 0)
- Un constructeur par copie.
- Un destructeur.

Opérateurs
- Une surcharge de l'opérateur = qui permettra de recopier des Tableaux. Lors de la copie la capacité sera la capacité minimale pour contenir une copie.
- Deux fonctions d'accès (lecture/écriture) aux données par surcharge de l'opérateur `[]`.

Opérations
- Une fonction **`void reserve(const int& nouvelle_capacite)`** qui permettra d'augmenter la capacité sans perte de données. Si la capacité proposée est plus petite que l'ancienne, aucune action n'est effectuée.
- Une fonction **`void retaille(const int& nouvelle_taille)`** qui change la taille des données utilisées. La capacité doit être augmentée au besoin.
- Une fonction **`void ajoute(const T& t)`** qui permettra d'ajouter une élément à la fin du tableau. La taille sera mise à jour et si la capacité du tableau n'est pas suffisante, elle sera augmentée (doubler la capacité est une bonne stratégie).
- Une fonction **`void supprime(const int& i)`** qui supprimera la case `i` du tableau. Pour cela, la dernière valeur contenue dans le tableau sera copiée en case `i` et la taille réduite de 1.

Accesseurs
- Une fonction retournant la taille des données stockées.
- Une fonction retournant la capacité

Exercice-3 :
On souhaiterait fournir une classe générique **`template class Handle<typename T>`**, qui permet de gérer les pointeurs alloués dynamiquement. Ce pointeur intelligent, aura la charge de détruire le pointeur alloué. Et toute copie de ce pointeur intelligent s'accompagnera de la duplication (*deep copy*) de l'objet pointé. On suppose pour cela que le type paramètre *Handle* impose un concept à son paramètre : celui de disposer d'une fonction de duplication *polymorphe* ou de clonage :
`T* clone()const;` (voir l'exemple ci-dessous).

Q-3-1 : Concevoir cette classe de sorte que le code suivante puisse fonctionner

Listing 7 – testHandle.cpp

```
#include <iostream>
```

```
#include "Handle.hpp"
//-------------------------------------------------------
class Label{
 int m_i;
 public:
 Label(int i=0):m_i(i){}
 Label(const Label& lab):m_i(lab.m_i){}
 ~Label(){std::cout<<"~Label()"<<std::endl;}
 Label* clone()const{return new Label(*this);}
 const int& label()const{return m_i;}
 int& label(){return m_i;}
 };
 //-------------------------------------------------------
 int main(int argc, char** argv){
 Handle<Label> pw1(new Label(2));
 Handle<Label> pw2 = pw1;
 (*pw2).label() = 3;
 std::cout << pw1->label() <<"\t"<< pw2->label()<<std::endl;
 return 0;
 }
```

Et afficher :

```
  2       3
~Label()
~Label()
```

Q-3-2 : Modifier (*si vous le pouvez*) votre conception de sorte que si le type T ne satisfait la contrainte d'offrir la fonction de clonage **`T* clone()const;`**, alors une erreur est signalée dès la compilation. (*Attention cette question est un peu technique, même pour les spécialistes.*)

Thème - 3 *Fonctions génériques : Applications*

Exercice-1 : Dans un problème, il est demandé de fournir une classe modélisant les fonctions de $\mathbb{R} \mapsto \mathbb{R}$, disposant de deux fonctions **`const`** renvoyant respectivement, pour une valeur réelle passée en paramètre, la valeur de la fonction et celle de sa dérivées (ou une valeur approximative par différences finies *voir précédent TP*). Le choix des noms de ces fonctions n'étant pas imposées, des classes sont fournies avec des noms différents pour ces fonctions membres :

Listing 8 – Concepteur A

```
class AFunction{
typedef double(*Fn)(double)
...
public:
void setFunc(Fn);

double f(double)const;
double df(double)const;
...
};
```

Listing 9 – Concepteur B

```
class CFunction{
typedef double(*Fn)(double)
...
public:
void setFunc(Fn);
double
operator()(double)const;
double d(double)const;
...
};
```

Q-1-1 : Fournir une fonction générique dont les paramètres seront à définir

```
double CFunctionEvaluate(const T& f, double x, double* df)
```

▲ qui prendra en arguments
- une instance **f** de type générique de la classe modélisant les fonctions réelles spécifiées ci-dessus.
- une valeur réelle **x**, identifiant le point d'évaluation de la fonction
- un pointeur **df**, qui lorsque fournie, devra stocker la valeur de la dérivée de **f** au point **x**

▲ et qui retournera la valeur de la fonction **f** au point **x**.

Q-1-2 : Prouver le bon fonctionnement de votre implémentation à l'aide d'une fonction test.

Thème - 4 *Métaprogrammation*

Exercice-1 : Ecrire une fonction paramétrique qui calcule le factoriel d'un entier pendant la compilation.

Q-1-1 : L'appel suivant devra donner le résultat.

Listing 10 – factoriel.cpp

```
#include <iostream>
#include "Factorial.hpp"
int main(){
std::cout<<" Factoriel de 50  est: "<< Factorial<50>::value <<"\n";
return 0;
}
```

Q-1-2 : Comparer le temps d'exécution à une approche récursive.

Exercice-2 : Ecrire une fonction paramétrique qui calcule la puissance d'un entier pendant la compilation.

Q-2-1 : Proposer une version lorsque la puissance et l'entier sont connue à la compilation.

Q-2-2 : Proposer une version lorsque seule la puissance est connue pendant la compilation.

Q-2-3 : Quelles différences faites vous des deux versions implémentées ?

Exercice-3 :
Concevoir une fonction générique **dot** qui effectue le produit scalaire de deux **TinyVector<T,N>** (*voir exercice 1 Thème 2*)

Q-3-1 : en déroulant et rendant en ligne (*inline*) la boucle,c'est-à-dire :

$$dot(u,v) \equiv u[0]*v[0]+u[1]*v[1]+\ldots+u[N-1]*v[N-1].$$

Q-3-2 : Quelle est l'avantage d'une telle fonction.